2020 IEEE 36th International Conference on Data Engineering (ICDE)

Continuously Tracking Core Items in Data Streams with Probabilistic Decays

Junzhou Zhao* Pinghui Wang†* Jing Tao* Shuo Zhang* John C.S. Lui‡
*MOEKLINNS Lab, School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, P. R. China
†Shenzhen Research Institute of Xi'an Jiaotong University, Shenzhen, P. R. China
‡Department of Computer Science & Engineering, The Chinese University of Hong Kong, Hong Kong
{junzhou.zhao, phwang, jtao}@xjtu.edu.cn zs412082986@stu.xjtu.edu.cn cslui@cse.cuhk.edu.hk

***Abstract*—The sheer scale of big data causes the information overload issue and there is an urgent need for tools that can draw valuable insights from massive data. This paper investigates the *core items tracking (CIT)* problem where the goal is to continuously track representative items, called *core items*, in a data stream so to best represent/summarize the stream. In order to simultaneously satisfy the *recency* and *continuity* requirements, we consider CIT over *probabilistic-decaying streams* where items in the stream are forgotten gradually in a probabilistic manner. We first introduce an algorithm, called PNDCIT, to find core items in a special kind of *probabilistic non-decaying streams*. Furthermore, using PNDCIT as a building block, we design two novel algorithms, namely PDCIT and PDCIT+, to maintain core items over probabilistic-decaying streams with constant approximation ratios. Finally, extensive experiments on real data demonstrate that PDCIT+ achieves a speedup of up to one order of magnitude over a batch algorithm while providing solutions with comparable quality.**

I. Introduction

In recent years, we have witnessed the rise of big data with an unprecedented scale of data being continuously generated from various sources such as social media, mobile devices, and sensor networks. Data streams such as tweet stream and news stream have become ubiquitous. However, their large volume and high velocity pose burdens for people to derive valuable information. For example, out of fear of missing out on important information, people tend to subscribe many information sources (e.g., follow many users on online social networks, subscribe many news outlets, etc), and consequently, they receive a lot of data containing redundant and noisy information, become overloaded, and effectively miss the information they are actually interested in. It is therefore imperative to develop tools to help people reduce data overload and draw valuable insights from massive data streams.

To solve this issue, one approach is to selectively maintain just a few representative items, called *core items*, to best represent/summarize the stream. The motivation can be illustrated by following examples. In a tweet stream, it is usually unnecessary to read all tweets, but selectively reading a few of them is enough to know the events happening recently [1,2]. Similarly, in a news stream, a few news articles may be enough to cover majority of the topics in the stream [3,4]. In a data point stream (e.g., check-in records) where points are distributed in an Euclidean space, a few centers are usually enough to characterize their distributions [5,6]. In these scenarios, we say that a few core items can represent/summarize the original massive data. The **challenge** is how to efficiently maintain core items in a *streaming fashion* where new items keep arriving, and the maintained core items should be updated accordingly to make sure that they are always (near-)optimal. We refer to this task as the *core items tracking* (CIT) task.

Pinghui Wang is the corresponding author.

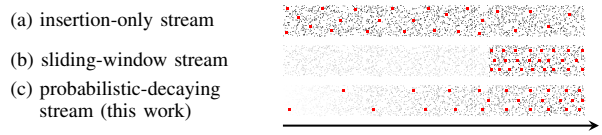


Fig. 1. Core items tracking over different stream models. A red dot represents a selected core item.

Specifically, CIT aims to maintain a set of core items that jointly maximize a prespecified *utility function* at any query time. For example, the utility function may measure the representativeness [2,7]–[9], informativeness [10]–[12], diversity [13], influence [14], or coverage [15,16], of a set of items. In many real world applications, utility functions are often found to have the *diminishing return* property, which can be captured by *submodular functions* [17]. Hence, CIT essentially requires solving the *streaming submodular optimization* (SSO) problem, which is NP-hard. Recently, several SSO techniques have been designed for *insertion-only streams* [3,7] and *sliding-window streams* [9,18,19], respectively. These SSO techniques can efficiently find core items with constant approximation ratios. However, the insertion-only stream and sliding-window stream that many existing SSO techniques are built upon, actually represent *two extremes*, and have their limitations.

In an insertion-only stream, core items are selected from all historical data items, which are treated with equal importance, regardless of how outdated they are. This is often undesirable because the stale historical data is usually less important than the fresh and recent data. In some scenarios, a large fraction of core items may be outdated items, as illustrated in Fig. 1(a). As a result, SSO techniques built on insertion-only streams may find core items that do not guarantee *recency*.

In a sliding-window stream, core items are selected from the

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00072

most recent data only, and historical data outside the window is completely discarded, as illustrated in Fig. 1(b). This may also be undesirable because one may not wish to completely lose the entire history of past data, for some historical data may be still important. Moreover, there is no golden rule on choosing a proper window size: a long window tends to find subsets containing many outdated items; a short window may find subsets containing many valueless/noisy items due to lack of data and hence result in rather unstable solutions. As a result, SSO techniques built on sliding-window streams may find core items that do not guarantee *continuity*.

In this work, we suggest that a better approach is to incorporate *temporal decaying* into core items selection. Specifically, we introduce a *probabilistic-decaying stream* (PDS) model that allows data items from the past to the present all have a chance to participate in analysis (hence satisfies continuity), but outdated data is less likely to participate in the analysis than recent data (hence satisfies recency). As time advances, an item will become increasingly less important, less likely to be selected as a core item, and hence the item will be gradually forgotten, as illustrated in Fig. 1(c).

Built on the PDS model, we design novel SSO techniques to solve the CIT problem with provable approximation guarantees and efficiency. We first consider CIT over a special kind of *probabilistic non-decaying* streams, and design an algorithm, called PNDCIT, to find core items over probabilistic non-decaying streams; PNDCIT guarantees an $(1/2-\epsilon)$ approximation ratio. Then, we show how to reduce the probabilistic-decaying case to the probabilistic non-decaying case, and leverage PNDCIT as a building block to design an algorithm, called PDCIT, to maintain core items in probabilistic-decaying streams; PDCIT also guarantees an $(1/2-\epsilon)$ approximation ratio. To improve the efficiency of PDCIT, we design an algorithm, called PDCIT+, which is fast and guarantees a $(1/4-\epsilon)$ approximation ratio. In summary, we make following contributions in this work.

- We propose the core items tracking problem over a more general probabilistic-decaying stream model. (Section II)
- We design a set of algorithms, from basic to advanced, to address the CIT problem with both update time and space efficiency. Importantly, our algorithms have provable constant approximation ratios. (Sections III–VI)
- We conduct extensive experiments on real data, and the results demonstrate that our method can find solutions with quality comparable with baseline methods, and could improve the computational efficiency to one order of magnitude faster. (Section VII)

II. Preliminaries

We first introduce some notations, and then formulate the core items tracking problem.

A. Notations

Data Stream. A data stream is an unbounded sequence of items arriving in chronological order. Each item e is from a set V, and e is associated with a discrete timestamp t_e. Multiple items may arrive at the same time, and there may be other attributes (e.g., age, gender, etc) associated with an item.

Utility Function. The utility function $f\colon 2^V \mapsto \mathbb{R}_{\geq 0}$ assigns each subset of V a nonnegative utility value, which could measure the representativeness, informativeness, diversity, influence, or coverage, of the subset. We shall focus on a special class of *monotone submodular functions* that are found to be ubiquitous in a wide range of applications [2,4,8,10,13,14,20].

Definition 1 (Monotone Submodular Function [17])**.** *For all $S \subseteq T \subseteq V$, a set function $f\colon 2^V \mapsto \mathbb{R}_{\geq 0}$ is monotone (non-decreasing) if $f(S) \leq f(T)$, and f is submodular if $f(S \cup \{e\}) - f(S) \geq f(T \cup \{e\}) - f(T), \forall e \in V \backslash T$.*

Let $\delta(e|S) \triangleq f(S \cup \{e\}) - f(S)$ denote the *marginal gain* of adding an item e to set S. Monotonicity implies that, adding an item to a set never decreases the set's utility, i.e., marginal gains are nonnegative $\delta(e|S) \geq 0$. Submodularity implies that, fixing item e, the item's marginal gain $\delta(e|S)$ never increases as set S grows larger, aka the *diminishing return* property. We also assume that f is *normalized*, i.e., $f(\emptyset) = 0$.

Our goal is to select a few items from the stream such that they jointly have the maximum utility. To clearly formulate this optimization problem, we still need a streaming model. We first review two existing streaming models, point out their limitations, and then propose a better model.

B. Limitations of Existing Data Stream Models

In the literature, two widely used data stream models are *insertion-only streams* [3,7] and *sliding-window streams* [9,18,19].

An insertion-only stream simply accumulates all the items in history and treats them with equal importance. A subset is then selected from a multiset $\mathcal{D}_t \triangleq \{e \in V\colon t_e \leq t\}$. Let $S(\mathcal{D}_t)$ denote the selected subset. As time advances to t', $S(\mathcal{D}_t)$ will be updated to $S(\mathcal{D}_{t'})$, either by computing it from scratch on $\mathcal{D}_{t'}$, or by incrementally updating the result from $S(\mathcal{D}_t)$ to $S(\mathcal{D}_{t'})$.

A sliding-window stream only keeps items in the most recent W time units and discards the others. A subset is selected from multiset $\mathcal{D}_{t-W,t} \triangleq \{e \in V\colon t_e \in [t-W, t]\}$ where W is the window size. Let $S(\mathcal{D}_{t-W,t})$ denote the selected subset at t. Similarly, as time advances to t', the output subset will be updated to $S(\mathcal{D}_{t'-W,t'})$ accordingly.

Note that both models actually take a *binary view* of an item, i.e., an item is either included for analysis (i.e., participates in subset selection) or not. An included item has the same level of importance as any other item, regardless of how outdated it is. As mentioned previously, this simplistic binary view makes it impossible to satisfy recency and continuity simultaneously. To address this limitation, we propose a better stream model that can ensure both recency and continuity.

C. Probabilistic-Decaying Stream Model

We propose a *probabilistic-decaying stream* (PDS) model to address the limitations of existing stream models. The PDS model enjoys a feature that, items from the past to the

present all have an opportunity to participate in analysis (hence satisfies continuity), however outdated items are less likely than recent items (hence satisfies recency). Therefore, PDS model can ensure recency and continuity simultaneously.

Formally, in the PDS model, at time t, each item $e \in \mathcal{D}_t$ independently participates in the analysis with probability $p(e,t) = h_e(t - t_e)$. Here, $h_e \colon \mathbb{Z}_{\geq 0} \mapsto [0,1]$ is an item-specific *decaying function* that assigns an item of a *certain age* with a probability of participating in the analysis, and $h_e(x)$ is a monotone non-increasing function. In practice, we require $h_e(x)$ be strictly decreasing, so an item in the distant history will have a less chance to participate in the analysis than a recent item, while outdated items are gradually forgotten.

The insertion-only and sliding-window stream models turn out to be two special cases of our PDS model: if $h_e(x) \equiv 1$ for all e with any age x, then a PDS becomes an insertion-only stream; if $h_e(x) = 1$ for $x \leq W$ and 0 otherwise for all e, then a PDS becomes a sliding-window stream.

D. Core Items Tracking over PDS

The PDS model allows each item in the stream to participate in the analysis with a decaying probability. The core items tracking problem then aims to choose a subset of items in the stream such that they jointly achieve the maximum utility *on average*. Formally, we have the following problem formulation.

Problem 1 (Core Items Tracking, CIT)**.** *Given a monotone submodular utility function f, a PDS with item-specific decaying function h_e, and a budget $k > 0$, we want to find a subset $S_t^* \subseteq V$ at any query time t such that*

$$S_t^* \in \underset{S \subseteq V \wedge |S| \leq k}{\arg\max} \; \mathbb{E}_{h_e}\left[f(S)|\mathcal{D}_t\right]. \tag{1}$$

The expectation is taken upon the randomness that each item participates in the analysis with a probability. We will omit subscript h_e if no confusion arises.

Note that CIT aims to maximize the *expected utility* on a PDS rather than the utility on a deterministic stream as is the case in previous work [7,9,18,19]. In the follows, we use a concrete example to explain how to evaluate $\mathbb{E}\left[f(S)|\mathcal{D}_t\right]$.

Example 1 (Online Maximum Coverage)**.** *Consider a PDS where each item e represents a set, e.g., topics in a news article, tags in a tweet, etc. Let $f(S) \triangleq |\cup_{e \in S} e|$ be the coverage of the set S. f is monotone and submodular. We want to choose at most $k = 2$ items to maximize the expected coverage in the PDS. For example, we may want to choose k news articles to cover the majority of topics in a probabilistic-decaying news stream. Fig. 2 gives a stream with 4 items arriving at $t = 1, 2, 3$, and 4. The last two columns give $p(e,t)$ at $t = 3$ and 4, respectively. The expected utility of subset $\{x, y\}$ at time t can be calculated by*

$$\begin{aligned}\mathbb{E}_{h_e}\left[f(\{x,y\})|\mathcal{D}_t\right] &= p(x,t)p(y,t)f(\{x,y\}) && (2)\\ &+ p(x,t)(1-p(y,t))f(\{x\}) && (3)\\ &+ (1-p(x,t))p(y,t)f(\{y\}). && (4)\end{aligned}$$

Here, (2) corresponds to the case that both x and y participate in analysis; (3) and (4) correspond to the cases that x or y participates in analysis. Observe that calculating $\mathbb{E}\left[f(S)|\mathcal{D}_t\right]$ requires $2^{|S|} - 1$ utility function evaluations.

We enumerate all the possible subsets with at most k items, calculate their expected utility, and find an optimal set at time $t = 3$ and 4, respectively, as shown in Fig. 2. Observe that as time advances from $t = 3$ to 4, item e_1's contribution is reduced as it becomes older.

t	item	$p(e,3)$	$p(e,4)$
1	$e_1 = \{a, b\}$	0.5	0.1
2	$e_2 = \{b\}$	0.8	0.5
3	$e_3 = \{c, d\}$	1.0	0.8
4	$e_4 = \{d, e\}$	-	1.0

At time $t = 3$:
$S_3^* = \{e_1, e_3\}$
expected utility: 3.0

At time $t = 4$:
$S_4^* = \{e_3, e_4\}$
expected utility: 2.8

Fig. 2. Probabilistic-decaying maximum coverage

In the follows, we will focus on a special class of *exponential decaying functions*, i.e., $h_e(x) = p_e^x$ where $p_e \in [0,1]$ relates to the item-specific decaying rate. The memoryless of exponential function can simplify our algorithm design (even though our framework can be extended to general decaying functions, see the full version of this paper).

Finally, it is worth noting that, in an insertion-only stream, budget k can be as large as the stream length $|\mathcal{D}_t|$, while in the PDS model, budget k is upper bounded by $\sum_{e \in \mathcal{D}_t} p(e,t)$. For example, in the exponential decaying case, if $p_e = p \in (0,1), \forall e$, and one item arrives at a time, then $k \leq 1/(1-p)$.

III. A Monte-Carlo Framework

As stated in Example 1, accurately calculating $\mathbb{E}\left[f(S)|\mathcal{D}_t\right]$ requires $O(2^{|S|})$ utility function evaluations, or we say $O(2^{|S|})$ *oracle calls*. For example, choosing $|S| = 100$ items needs 2^{100} oracle calls, which is too expensive. This section proposes a Monte-Carlo simulation framework to estimate $\mathbb{E}\left[f(S)|\mathcal{D}_t\right]$.

Given data stream $\mathcal{D}_t$ and decaying function h_e, we define a *realization* of the PDS as a multiset of *active items* that indeed participated in analysis, and there are $2^{|\mathcal{D}_t|}$ possible realizations, denoted by $\mathbb{D}_t \triangleq \{\mathcal{D} \colon \mathcal{D} \subseteq \mathcal{D}_t\}$. Each realization $\mathcal{D} \in \mathbb{D}_t$ is observed with probability

$$P(\mathcal{D} \in \mathbb{D}_t) = \prod_{e \in \mathcal{D}_t} p(e,t)^{\mathbf{1}(e \in \mathcal{D})}(1 - p(e,t))^{\mathbf{1}(e \notin \mathcal{D})} \tag{5}$$

where $\mathbf{1}(\cdot)$ denotes the indicator function. The Monte-Carlo method [21] states that, a collection of samples $\{\mathcal{D}_t^{(i)}\}_{i=1}^n$ independently drawn from distribution $\{P(\mathcal{D}) \colon \mathcal{D} \in \mathbb{D}_t\}$ can provide an unbiased estimate of expectation $\mathbb{E}\left[f(S)|\mathcal{D}_t\right]$. Specifically, we have the following lemma.

Lemma 1. *Let $\{\mathcal{D}_t^{(i)}\}_{i=1}^n$ be a set of n samples independently drawn from distribution $\{P(\mathcal{D}) \colon \mathcal{D} \in \mathbb{D}_t\}$. Then,*

$$F(S) \triangleq \frac{1}{n}\sum_{i=1}^{n} f(S \cap \mathcal{D}_t^{(i)}) \xrightarrow{a.s.} \mathbb{E}\left[f(S)|\mathcal{D}_t\right], \quad n \to \infty \tag{6}$$

where $\xrightarrow{a.s.}$ denotes almost sure convergence.

Intuitively, a subset S will have large expected utility if S contains many active items, i.e., $|S \cap \mathcal{D}_t^{(i)}|$ is large, and they have large utilities on average. Fig. 3 shows some data stream samples.

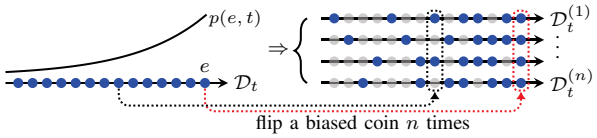


Fig. 3. Data stream samples. Solid dots denote active items that participate in analysis. Grayed dots denote inactive items that do not participate in analysis.

The main advantage of Monte-Carlo framework is that the number of oracle calls for evaluating expected utility decreases from $O(2^{|S|})$ to $O(n)$. Sample size n can be determined by applying the Hoeffding's inequality, and usually $n \ll 2^{|S|}$.

Lemma 2. *Let $\{\mathcal{D}_t^{(i)}\}_{i=1}^n$ be a collection of n samples independently drawn from distribution $\{P(\mathcal{D})\colon \mathcal{D} \in \mathbb{D}_t\}$. If $n \geq \frac{\ln(2/\delta)}{2\epsilon^2\rho^2}$ where $\epsilon, \delta \in (0,1)$ and $\rho \triangleq \frac{\mathbb{E}[f(S)|\mathcal{D}_t]}{f(S)}$, then*

$$\left|F(S) - \mathbb{E}\left[f(S)|\mathcal{D}_t\right]\right| < \epsilon \mathbb{E}\left[f(S)|\mathcal{D}_t\right]$$

holds with probability at least $1-\delta$.

In the follows, we will assume that n is sufficiently large so that $F(S) \approx \mathbb{E}\left[f(S)|\mathcal{D}_t\right], \forall S \subseteq V$.

In Eq. (6), given $\mathcal{D}_t^{(i)}$, $f(S \cap \mathcal{D}_t^{(i)})$ is a monotone submodular function of S, so $F(S)$ is monotone and submodular because monotonicity and submodularity are closed under positive linear combinations. A straightforward way to solve CIT is to run a GREEDY algorithm [17] on those active items to maximize F. However, GREEDY is not a streaming algorithm, and it is not fast (more will be discussed in Section VII).

The CIT problem boils down to two sub-problems: (1) how to efficiently maintain data stream samples $\{\mathcal{D}_t^{(i)}\}_{i=1}^n$, and (2) how to design a streaming algorithm to maximize F. We address these two sub-problems in the remainder of this paper.

IV. Maintaining Data Stream Samples

In this section, we address the sub-problem of maintaining data stream samples $\{\mathcal{D}_t^{(i)}\}_{i=1}^n$.

A. Sampling Methods

Naive Sampling. A straightforward way to obtain data stream samples $\{\mathcal{D}_t^{(i)}\}_{i=1}^n$ is to conduct Bernoulli sampling on $\mathcal{D}_t$ from scratch at each time t. That is, for each item $e \in \mathcal{D}_t$, we flip a biased coin (with probability $p(e,t)$ of heads) n times. If the i-th trial succeeds (say, the outcome is a head), then e is included in $\mathcal{D}_t^{(i)}$; otherwise e is not included in $\mathcal{D}_t^{(i)}$, as illustrated in Fig. 3. Obviously, this approach is inefficient.

Incremental Sampling. Given $h_e(x) = p_e^x$, we can construct $\mathcal{D}_t^{(i)}$ from $\mathcal{D}_{t-1}^{(i)}$ incrementally. In more detail, each new item arrived at time t is always included in $\mathcal{D}_t^{(i)}$; each existing item $e \in \mathcal{D}_{t-1}^{(i)}$ is included in $\mathcal{D}_t^{(i)}$ with probability p_e. It is easy to see that the resulting $\mathcal{D}_t^{(i)}$ follows the desired distribution. Incremental sampling should be more efficient than naive sampling. However, conducting Bernoulli sampling for each existing item at each time step is still time consuming.

Lifetime Sampling. Incremental sampling exhibits a feature that, an item is always included when it arrives, then it survives for an amount of time, and finally it is discarded. This observation allows us to think that each item e has a *lifetime* l_e, which is the time span from its arrival till being discarded. Given $h_e(x) = p_e^x$, we find that l_e actually follows the *geometric distribution* $\mathrm{Geo}(p_e)$, i.e., $P(l_e = l) = p_e^{l-1}(1-p_e), l = 1, 2, \ldots$. Leveraging this observation, for item e arrived at time t_e, we only need to sample n lifetimes $\{l_e^{(i)}\}_{i=1}^n$ from $\mathrm{Geo}(p_e)$. Let lifetimes decrease over time: for item e, as time advances to t, its lifetime decreases to $l_e^{(i)}(t) = l_e^{(i)} - (t - t_e), i = 1, \ldots, n$. If $l_e^{(i)}(t)$ becomes zero, e is discarded from $\mathcal{D}_t^{(i)}$. At time t, $\mathcal{D}_t^{(i)}$ simply consists of all items with non-zero lifetimes, i.e., $\mathcal{D}_t^{(i)} = \{e \in \mathcal{D}_t\colon l_e^{(i)}(t) > 0\}$. Lifetime sampling only needs to perform n sampling for each item at its arrival time. Hence, it is faster than incremental sampling.

In essence, above three sampling methods are equivalent to each other, and they help us understand data stream sampling from different perspectives.

B. Bernoulli Set and the Shrinking Property

Before moving to solve the second sub-problem, we introduce the notation of *Bernoulli set*, which will be useful in later discussions. An item participates in analysis if it is included in at least one of these n data stream samples. In other words, an item participates in analysis if at least one of its n Bernoulli trials succeeded (recall the naive sampling). Let

$$I(e) \triangleq \{i\colon i\text{-th Bernoulli trial of } e \text{ succeeds}\} \tag{7}$$

denote the succeeded Bernoulli trials of item e. We refer to $I(e)$ as item e's *Bernoulli set*. As time advances, an item has less chance to participate in analysis. This is reflected by the fact that an item's Bernoulli set is *shrinking* over time. If an item's Bernoulli set becomes empty at some time, the item no longer participates in analysis. Therefore, shrinking Bernoulli set is another way to understand the decaying nature in PDS.

From the perspective of lifetime sampling, we can conveniently write item e's Bernoulli set at time $t = t_e + l$ (i.e., l time units after its arrival) by

$$I_l(e) \triangleq \{i\colon l_e^{(i)}(t) > 0\} = \{i\colon l_e^{(i)} > l\}, \quad l = 0, 1, \ldots \tag{8}$$

and $I_l(e) = \emptyset$ when $l \geq \max_i l_e^{(i)}$, as illustrated in Fig. 4.

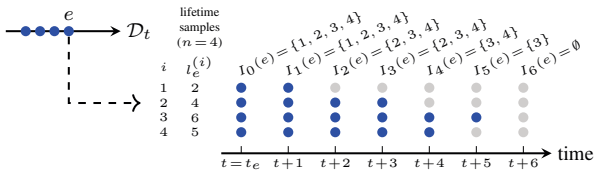


Fig. 4. Shrinking Bernoulli sets

V. A Basic Streaming Algorithm

In this section, we design a basic streaming algorithm, called PDCIT, to solve the CIT problem. We will first consider a special *probabilistic non-decaying* case, and design PNDCIT to solve CIT over this special case. PNDCIT is used as a building block to design PDCIT to solve the CIT problem.

A. The Probabilistic Non-Decaying Case

Consider a special case where each item's decaying function is a constant, i.e., $h_e(x) \equiv p_e \in [0,1], \forall e$. So $p(e,t) \equiv p_e$. In other words, each item participates in analysis with a constant probability, referred to as the *probabilistic non-decaying case* (see Fig. 5). Solving CIT over this special case is perhaps not interesting per se, but the algorithm to solve it will be a key ingredient for solving the general CIT problem.

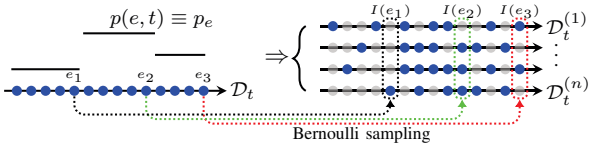


Fig. 5. The probabilistic non-decaying case

Since $p(e,t)$ is time-invariant, e's Bernoulli set is a constant, denoted by $I(e)$. In other words, $I(e)$ does not shrink in the probabilistic non-decaying case, see Fig. 5. It turns out that the PDS in this special case can be viewed as an insertion-only stream where *each item is a non-shrinking Bernoulli set*, i.e., PDS $\{e_1, e_2, \ldots\}$ is viewed as an insertion-only stream $\{I(e_1), I(e_2), \ldots\}$. SSO over insertion-only streams has been extensively studied [3,7], and these algorithms can be easily adapted to our case. The main operation in these algorithms is to compute the marginal gain of a new item with respect to currently chosen items; if the gain is sufficiently large, the new item is chosen; otherwise, it is dropped. In our case, the marginal gain can be calculated by

$$\Delta(e|S) \triangleq F(S \cup \{e\}) - F(S) = \frac{1}{n} \sum_{i \in I(e)} \delta(e|S \cap \mathcal{D}_t^{(i)}). \quad (9)$$

We have adapted the state-of-the-art SieveStreaming algorithm [7] to our case, denoted by PNDCIT, with pseudo-code given in Alg. 1. Similar to SieveStreaming, PNDCIT ensures the following properties.

Alg. 1: PNDCIT (adapted from SieveStreaming [7])

Input: A stream of Bernoulli sets $\{I(e): e \in \mathcal{D}_t\}$
Output: Core items $S_t \subseteq V$ at time t

$\Delta_m \leftarrow 0, \Theta \leftarrow \emptyset$, $\{S_\theta\}_{\theta \in \Theta}$ is a family of candidate sets;
foreach *Bernoulli set* $I(e)$ *in the stream* **do**
  $\Delta_m \leftarrow \max\{\Delta_m, \frac{|I(e)|}{n} f(\{e\})\}$;
  $\Theta' \leftarrow \{(1+\epsilon)^i : \frac{\Delta_m}{2k} \le (1+\epsilon)^i \le \Delta_m, i \in \mathbb{Z}\}$;
  Delete S_θ for $\theta \in \Theta \backslash \Theta'$ and let $S_\theta \leftarrow \emptyset$ for $\theta \in \Theta' \backslash \Theta$;
  $\Theta \leftarrow \Theta'$;
  foreach *threshold* $\theta \in \Theta$ **do**
    if $|S_\theta| < k$ *and* $\Delta(e|S_\theta) \ge \theta$ **then** $S_\theta \leftarrow S_\theta \cup \{e\}$;
$S_t \leftarrow S_{\theta^*}, \theta^* = \arg\max_{\theta \in \Theta} F(S_\theta)$;

Theorem 1. *PNDCIT achieves an* $(1/2 - \epsilon)$ *approximation ratio.*

Theorem 2. *PNDCIT requires* $O(n\epsilon^{-1} \log k)$ *oracle calls to process each item and* $O(nk\epsilon^{-1} \log k)$ *space to store intermedia results.*

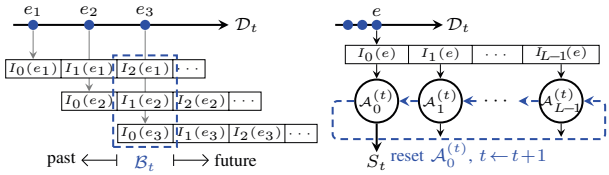
Remark. PNDCIT will be served as a blackbox: when fed with a stream of constant Bernoulli sets, PNDCIT outputs an $(1/2 - \epsilon)$-approximate solution of the CIT problem in the probabilistic non-decaying case. This viewpoint will be important in following discussions.

B. The PDCIT Algorithm

Now consider the case $h_e(x) = p_e^x$. In this case, the probability that each item participates in analysis decreases over time. In other words, each item's Bernoulli set is shrinking (see Fig. 4). We show that the probabilistic decaying case can be reduced to probabilistic non-decaying case. So we can leverage PNDCIT as a building block to address the CIT problem.

Recall that item e's Bernoulli set is $I_0(e)$ at time t_e, then shrinks to $I_1(e)$ at time $t_e + 1$, and to $I_l(e)$ at time $t_e + l$. **Assume lifetime is upper bounded by** L. Then e's Bernoulli set must become empty before $t_e + L$, i.e., $I_l(e) = \emptyset$ for $l \ge L$. At time t, each active item in the stream corresponds to a nonempty Bernoulli set. Let $\mathcal{B}_t \triangleq \{I_l(e) \ne \emptyset : e \in \mathcal{D}_t \wedge t_e + l = t\}$ be a family of nonempty Bernoulli sets at t (cf. Fig. 6(a)).

Ideally, if we can feed $\mathcal{B}_t$ to a PNDCIT instance, then the instance's output will be an $(1/2-\epsilon)$-approximate solution at t. The challenge is that, $\mathcal{B}_t$ keeps changing: each active item's Bernoulli set in $\mathcal{B}_t$ is shrinking, removed when becoming empty, and meanwhile, Bernoulli sets of newly arrived items at t are added into $\mathcal{B}_t$. How should one process $\mathcal{B}_t$ using PNDCIT in a streaming fashion?



(a) Bernoulli sets at t (b) PDCIT

Fig. 6. Bernoulli sets at t and PDCIT

We propose PDCIT to address this challenge. PDCIT runs L PNDCIT instances in parallel at any time t, denoted by $\{\mathcal{A}_l^{(t)}\}_{l=0}^{L-1}$. PDCIT processes items in the stream as follows.

• **Processing.** When a new item e arrives, we use lifetime sampling to obtain its (at most) L Bernoulli sets $\{I_l(e)\}_{l=0}^{L-1}$. PDCIT then *proactively* processes these L Bernoulli sets by L PNDCIT instances in parallel (cf. Fig. 6(b)). After items at t all have been processed, $\mathcal{A}_0^{(t)}$ outputs the solution at t.

• **Shifting.** Before processing the upcoming items at $t+1$, PDCIT first resets $\mathcal{A}_0^{(t)}$, and *circularly shifts* these L instances one unit to the left. (Each instance's subscript is also modified so the first instance always starts from 0, see Fig. 6(b)). Then,

it keeps on processing new items arrived at $t+1$ similarly as at t. The output of the first instance is the solution at $t+1$.

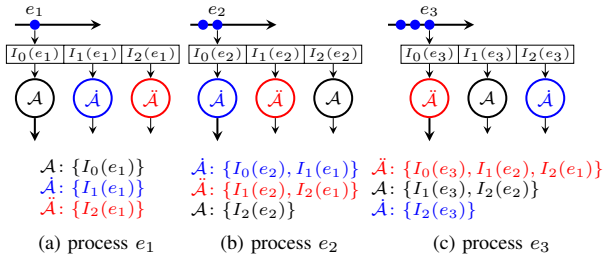
PDCIT repeats the above procedure, and continuously processes items in the data stream. Readers may find the following example helpful in understanding the execution of PDCIT.

Example 2. *PDCIT processes the stream $\{e_1, e_2, e_3\}$ in Fig. 6(a) as follows. Assume items' lifetimes are upper bounded by $L = 3$. Then PDCIT maintains three PNDCIT instances, denoted by $\mathcal{A}, \dot{\mathcal{A}}$ and $\ddot{\mathcal{A}}$.*

When e_1 arrives, its Bernoulli sets $\{I_l(e_1)\}_{l=0}^{2}$ are generated and fed to $\mathcal{A}, \dot{\mathcal{A}}$ and $\ddot{\mathcal{A}}$, respectively, see Fig. 7(a).

Before processing e_2, we reset $\mathcal{A}$, circularly shift $\mathcal{A}, \dot{\mathcal{A}}$ and $\ddot{\mathcal{A}}$ one unit to the left. Then e_2's Bernoulli sets $\{I_l(e_2)\}_{l=0}^{2}$ are generated and fed to $\dot{\mathcal{A}}, \ddot{\mathcal{A}}$ and $\mathcal{A}$, respectively, see Fig. 7(b).

Item e_3 is processed similarly, see Fig. 7(c). Note that, at any time, the first instance always correctly processed all the Bernoulli sets in the stream, i.e., $\mathcal{B}_t$.



(a) process e_1 (b) process e_2 (c) process e_3

Fig. 7. PDCIT example. Bottom shows the processed Bernoulli sets of each instance. After processing e_3, the first instance $\ddot{\mathcal{A}}$ processed all the Bernoulli sets, i.e., $\mathcal{B}_t$ in Fig. 6(a).

PDCIT ensures that the first PNDCIT instance $\mathcal{A}_0^{(t)}$ always processed all the Bernoulli sets in family $\mathcal{B}_t$ at any time t. Hence $\mathcal{A}_0^{(t)}$'s output is the solution at t, and guarantees an $(1/2-\epsilon)$ approximation ratio. The pseudo-code of PDCIT is given in Alg. 2. PDCIT has following properties.

Alg. 2: PDCIT

```
Initialize L PNDCIT instances {A_l^(1)}_{l=0}^{L-1};
foreach item e arrived at time t = 1, 2, ... do
    Obtain e's Bernoulli sets {I_l(e)}_{l=0}^{L-1};        // Processing
    for l ∈ [0, L − 1] do Feed A_l^(t) with I_l(e);
    S_t ← output of A_0^(t);
    for l = 1, ..., L − 1 do A_{l−1}^(t+1) ← A_l^(t);      // Shifting
    Reset A_0^(t) and A_{L−1}^(t+1) ← A_0^(t);
```

Theorem 3. *PDCIT achieves an $(1/2-\epsilon)$ approximation ratio.*

Theorem 4. *PDCIT requires $O(Ln\epsilon^{-1}\log k)$ oracle calls to process each item and $O(Lnk\epsilon^{-1}\log k)$ space to store intermedia results.*

As PNDCIT instances in PDCIT are independent with each other, they can be executed in parallel. Hence, the scalability of PDCIT can be further improved.

PDCIT is suitable for the case that lifetime upper bound L is small. This occurs when items' participation probabilities decrease quickly, i.e., p_e's are small. However, if items' participation probabilities decrease slowly, i.e., p_e's are large, L will be large. Maintaining a large number of PNDCIT instances will incur high CPU and RAM overloads, and make PDCIT inefficient. Therefore, processing items with slowly decaying rates is the main **bottleneck** of PDCIT. We address this limitation in next section.

VI. A Faster Streaming Algorithm

In this section, we address the bottleneck of PDCIT by designing a faster streaming algorithm, called PDCIT+, which allows lifetime upper bound L to be infinitely large.

A. Basic Idea

Instead of maintaining L PNDCIT instances in PDCIT, our idea is to *selectively maintain just a few of them*, and hope these selected instances can well approximate the rest. Because only a few instances are maintained, efficiency should be improved a lot. This idea can be thought of as using a histogram to approximate a curve.

More precisely, let $g_t(l)$ denote the utility value of instance $\mathcal{A}_l^{(t)}$'s output at time t, i.e., $g_t(l) \triangleq F(S_{t,l})$ where $S_{t,l} \subseteq V$ is $\mathcal{A}_l^{(t)}$'s output. We want to selectively maintain a few indices $\mathbf{x}_t \triangleq \{x_1, x_2, \ldots\} \subseteq \mathbb{Z}_{\geq 0}$ so that histogram $\{g_t(l)\}_{l\in\mathbf{x}_t}$ can well approximate "curve" $\{g_t(l)\}_{l\geq 0}$ at any time t (cf. Fig. 8). If the size of $\mathbf{x}_t$ is indeed small, then PDCIT+ will be much faster than PDCIT. Similar to PDCIT, in PDCIT+, the output of the first instance, i.e., $\mathcal{A}_{x_1}^{(t)}$, will be the solution at t.

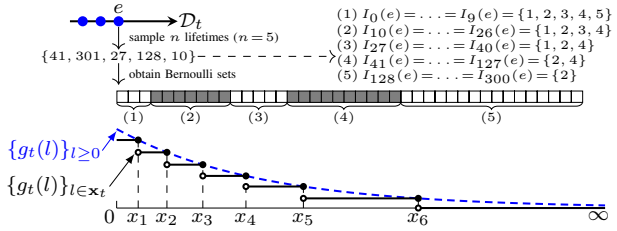


Fig. 8. Approximating "curve" $\{g_t(l)\}_{l\geq 0}$ by histogram $\{g_t(l)\}_{l\in\mathbf{x}_t}$. Note that curve $\{g_t(l)\}_{l\geq 0}$ does not have to be monotone.

The challenge is how to update index set $\mathbf{x}_t$ when new items arrive so that $\mathcal{A}_{x_1}^{(t)}$'s output is always close to the optimal solution at any time t.

B. Updating the Index Set

Before elaborating PDCIT+, we first give a useful observation about an item's Bernoulli sets. Recall that in PDCIT, when an item e arrives, we obtain its L Bernoulli sets $\{I_l(e)\}_{l=0}^{L-1}$, and then feed them to L PNDCIT instances, respectively. When Bernoulli sets are generated using n lifetime samples, there are actually at most n distinct Bernoulli sets. Consecutive Bernoulli sets are often the same, and they can be grouped into one *segment*. There will be at most n segments for each item, and each segment corresponds to a unique Bernoulli set.

For example, in Fig. 8, after sampling $n = 5$ lifetimes for item e, we find that $I_0(e)$ to $I_9(e)$ are the same, and they form the first segment; $I_{10}(e)$ to $I_{26}(e)$ form the second segment, and so on. There are totally five segments for item e.

With above observation, there is an equivalent way to execute PDCIT. Let $\langle l_i, r_i \rangle$ be the i-th segment of e's Bernoulli sets, and $I_{l_i}(e) = I_{l_i+1}(e) = \ldots = I_{r_i}(e)$.

If e is the first item in the stream, then after processing e by PDCIT, we only need to maintain PNDCIT instances indexed by $\mathbf{x}_t = \{r_i \colon 1 \leq i \leq n\}$ as instances indexed by $\{l_i, l_i + 1, \ldots, r_i\}$ all have the same outputs.

When new items keep arriving, we can create new PNDCIT instances (when needed) just based on currently maintained instances, and ensure that the results are always the same as PDCIT. For each segment $\langle l, r \rangle$ of a newly arrived item at t, we check whether instance $\mathcal{A}_r^{(t)}$ exists. If not exists, we create $\mathcal{A}_r^{(t)}$ as follows: if r has a successor $i \in \mathbf{x}_t$, then $\mathcal{A}_r^{(t)}$ is a clone of $\mathcal{A}_i^{(t)}$; otherwise $\mathcal{A}_r^{(t)}$ is newly created. We save r in $\mathbf{x}_t$, and feed the segment's Bernoulli set to each $\mathcal{A}_j^{(t)}, j \in [l, r]$.

For example, in Fig. 8, consider processing the second segment $\langle 10, 26 \rangle$ of e. We need to create a new instance $\mathcal{A}_{26}^{(t)}$ by cloning $\mathcal{A}_{x_4}^{(t)}$, and feed the segment's Bernoulli set $\{1, 2, 3, 4\}$ to $\mathcal{A}_{x_2}^{(t)}$, $\mathcal{A}_{x_3}^{(t)}$ and $\mathcal{A}_{26}^{(t)}$, respectively.

Above procedure continues, and we are actually running PDCIT in another equivalent way. However, as new items keep arriving, index set $\mathbf{x}_t$ will grow increasingly large, and we have to maintain a large number of PNDCIT instances, resulting in poor efficiency. To address this issue, we propose to reduce the number of running instances by removing *redundant* ones.

C. Reducing Redundancy

Intuitively, if several PNDCIT instances have very close outputs, it is not necessary to maintain all of them, as some of them may be redundant, and can be killed. We formally define *redundancy* as follows.

Definition 2 (ϵ-redundancy)**.** *Consider two instances $\mathcal{A}_i^{(t)}$ and $\mathcal{A}_l^{(t)}$ with $i < l$. We say $\mathcal{A}_l^{(t)}$ is ϵ-redundant if there exists $j > l$ such that $g_t(j) \geq (1 - \epsilon) g_t(i)$.*

That said, since $\mathcal{A}_i^{(t)}$ and $\mathcal{A}_j^{(t)}$ are already close with each other, then instances between them are redundant. In PDCIT+, we regularly check the output of each PNDCIT instance, and kill those redundant ones. PDCIT+ is described in Alg. 3.

The only issue of PDCIT+ is that, when we create $\mathcal{A}_r^{(t)}$ by cloning its successor $\mathcal{A}_i^{(t)}$, if some instances with indices between r and i were killed before, then $\mathcal{A}_i^{(t)}$ is actually not the "true" successor of $\mathcal{A}_r^{(t)}$. As a result, $\mathcal{A}_r^{(t)}$ may not process all the Bernoulli sets that it should process, and results in poor solution quality. Here, we postpone describing the solution to this issue, and assume there is an "oracle" that can correctly restore $\mathcal{A}_r^{(t)}$, as described in Line 14 of Alg. 3. We later discuss how to implement this oracle.

D. Theoretical Analysis

We show that PDCIT+ guarantees a constant approximation ratio. Proofs are in the extended version of this paper.

Alg. 3: PDCIT+

```
1  Initialize index set x_1 ← ∅;
2  foreach item e arrived at time t = 1, 2, ... do
3      Obtain e's segments of Bernoulli sets;
4      foreach segment ⟨l, r⟩ do Process(⟨l, r⟩);
5      S_t ← output of A_{x_1}^{(t)};
6      if x_1 = 0 then
7          Kill the outdated instance A_{x_1}^{(t)};
8          x_t ← x_t \ {x_1};
9      for i = 1, ..., |x_t| do
10         A_{x_i - 1}^{(t+1)} ← A_{x_i}^{(t)}, x_i^{(t+1)} ← x_i^{(t)} − 1;
11 Procedure Process(⟨l, r⟩)
12     if r ∉ x_t then
13         if r has no successor then Create a new A_r^{(t)};
14         else Restore A_r^{(t)} via an oracle;
15         x_t ← x_t ∪ {r};
16     Feed the segment's Bernoulli set to A_i^{(t)}, i ∈ [l, r];
17     ReduceRedundancy();
18 Procedure ReduceRedundancy()
19     foreach i ∈ x_t do
20         Find the largest j > i s.t. g_t(j) ≥ (1 − ε) g_t(i);
21         Delete each l ∈ (i, j) from x_t and kill A_l^{(t)};
```

Due to the shift operation in PDCIT+, indices $x \in \mathbf{x}_t$ and $x + 1 \in \mathbf{x}_{t-1}$ actually index the same PNDCIT instance. We say $x' \in \mathbf{x}_{t'}$ is an *ancestor* of $x \in \mathbf{x}_t$ if $t' \leq t$ and $x' = x + t - t'$. In the follows, we will always use symbol x' to denote the ancestor of x at time $t' \leq t$.

Let $\mathcal{A}$ be a PNDCIT instance. Let $\mathcal{A}(\mathcal{B})$ be $\mathcal{A}$'s output utility after processing Bernoulli set stream $\mathcal{B}$. Let $\mathcal{B}||\mathcal{B}_1$ be the concatenation of two Bernoulli set streams $\mathcal{B}$ and $\mathcal{B}_1$.

Lemma 3. *PNDCIT is suffix monotone, i.e., $\mathcal{A}(\mathcal{B}||\mathcal{B}_1) \geq \mathcal{A}(\mathcal{B}), \forall \mathcal{B}_1$.*

Lemma 4. *Let $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ be three Bernoulli set streams. The j-th Bernoulli set of $\mathcal{B}_i$ is $I_{ij}(e_j), e_j \in V$, and $I_{1j}(e_j) \supseteq I_{2j}(e_j) \supseteq I_{3j}(e_j)$. Let $\mathrm{OPT}(\mathcal{B})$ be the value of an optimal solution in $\mathcal{B}$. If $\mathcal{A}(\mathcal{B}_3) \geq (1 - \epsilon)\mathcal{A}(\mathcal{B}_1)$, then $\mathcal{A}(\mathcal{B}_3||\mathcal{B}) \geq (1/4 - \epsilon)\mathrm{OPT}(\mathcal{B}_2||\mathcal{B}), \forall \mathcal{B}$.*

Lemma 4 states a scene that, once PNDCIT on $\mathcal{B}_3$ and $\mathcal{B}_1$ have close outputs, then PNDCIT on $\mathcal{B}_3||\mathcal{B}$ will find a near optimal solution in $\mathcal{B}_2||\mathcal{B}$, as long as the input Bernoulli sets meet the condition in the lemma.

Lemma 5. *For two consecutive indices $x_i, x_{i+1} \in \mathbf{x}_t$, one of the following two cases must hold:*

Case 1 *No index is created between these two indices (and their ancestors) ever since they were created. In other words, no item has a segment $\langle l, r \rangle$ with $x'_i < r < x'_{i+1}$ since the two indices were created.*

Case 2 *At some time $t' \leq t$, it holds that $g_{t'}(x'_{i+1}) \geq (1 - \epsilon) g_{t'}(x'_i)$, and from t' to t, no index is ever created between these two indices.*

Lemma 5 is actually straightforward and it states the only two possible reasons why two indices in $\mathbf{x}_t$ are consecutive: either because they are consecutive since they were created (Case 1), or they become consecutive after redundant instances

between them were removed (Case 2). Leveraging Lemmas 4 and 5, we now obtain the main result of PDCIT+.

Theorem 5. *PDCIT+ achieves a* $(1/4 - \epsilon)$ *approximation ratio.*

Theorem 6. *PDCIT+ requires* $O(n\epsilon^{-2}\log^2 k)$ *oracle calls to process each item and* $O(nk\epsilon^{-2}\log^2 k)$ *space to store intermedia results.*

PDCIT+ uses a histogram to approximate a curve, hence it has a weaker approximation ratio than PDCIT. In experiments, we find that PDCIT+ actually finds solutions with quality very close to PDCIT, and it is much faster.

E. PDCIT+ Implementation

PDCIT+ relies on an oracle that can restore $\mathcal{A}_r^{(t)}$ when a new item with segment $\langle l, r\rangle$ arrives and $r \notin \mathbf{x}_t$ (Line 14 in Alg. 3). A naive way to implement this oracle is that: when an instance is killed, we swap it out of RAM; then we store the unprocessed Bernoulli sets corresponding to each killed instance (in RAM or on disk); when $\mathcal{A}_r^{(t)}$ needs to be restored, we swap it in RAM again and feed it with the corresponding unprocessed Bernoulli sets. However, frequently swapping in/out of RAM and storing unprocessed Bernoulli sets will harm the efficiency a lot. Instead, we propose another more efficient way to implement PDCIT+.

The key idea is as follows. We do not wish to accurately restore $\mathcal{A}_r^{(t)}$, but allow restoring with some tolerable error. As compensation, we delete redundant instances more conservatively, and slightly more instances will be kept.

A redundant instance is killed because the instance has similar outputs with its successor. We still restore $\mathcal{A}_r^{(t)}$ from its successor $\mathcal{A}_j^{(t)}$ (i.e., $\mathcal{A}_r^{(t)}$ is a clone of $\mathcal{A}_j^{(t)}$). If we do not feed $\mathcal{A}_r^{(t)}$ with the unprocessed historical Bernoulli sets, there will be a gap between its actual output $\hat{g}_t(r)$ and expected output $g_t(r)$. We only need to worry about the case $\hat{g}_t(r) < g_t(r)$, as the other case $\hat{g}_t(r) \geq g_t(r)$ means that without processing historical Bernoulli sets, $\mathcal{A}_r^{(t)}$ finds even better solutions (which may rarely happen in practice but indeed possible).

Because $g_t(r)$ is unknown, to avoid removing instances that are actually not redundant, we give each instance $\mathcal{A}_r^{(t)}$ an amount of *uncertainty*, denoted by δ_r, as compensation for not processing all Bernoulli sets ($g_t(r)$ may be as large as $\hat{g}_t(r) + \delta_r$). This allows us to represent $g_t(r)$ by an interval $[\underline{g}_t(r), \overline{g}_t(r)]$ where $\underline{g}_t(r) \triangleq \hat{g}_t(r)$ and $\overline{g}_t(r) \triangleq \hat{g}_t(r) + \delta_r$. As $\underline{g}_t(j) \geq (1-\epsilon)\overline{g}_t(i)$ implies $g_t(j) \geq (1-\epsilon)g_t(i)$, the redundancy condition is thus relaxed to $\underline{g}_t(j) \geq (1-\epsilon)\overline{g}_t(i)$.

We want uncertainty δ_r to be related to the amount of historical Bernoulli sets that $\mathcal{A}_r^{(t)}$ does not process. Notice that if a redundant instance indexed by r is removed in interval (i, j) at time t, we can approximate this uncertainty by $g_t(i) - g_t(j) \leq \epsilon g_t(i)$. That is, we set $\delta_r = \epsilon\overline{g}_t(i)$.

The final implementation of PDCIT+ is given in Alg. 4. We will verify its performance in experiments.

Alg. 4: PDCIT+ (final implementation)

```
1  Procedure Process(⟨l, r⟩)
2      if r ∉ x_t then
           // ...
3          if r has a successor then
4              Let i and j denote r's predecessor (or i = 0 if not
                 exists) and successor, respectively;
5              A_r^(t) ← a copy of A_j^(t);
6              Find a, b ∈ x_t s.t. (a, b) ⊇ (i, j) and δ_ab is recorded,
                 let δ_r ← δ_ab;
           // ...
7  Procedure ReduceRedundancy()
8      foreach i ∈ x_t do
9          Find the largest j > i s.t. g_t(j) ≥ (1 − ε)ḡ_t(i);
10         Delete index l ∈ (i, j) from x_t and kill A_l^(t);
11         δ_ij ← εḡ_t(i);
```

VII. Experiments

In this section, we use public available real data sets to validate our algorithms. We will study two special cases of the CIT problem: the *representative item selection* problem and the *Gaussian process active learning* problem.

A. Datasets

DBLP [22]. The dataset records the meta information of about 3 million papers, including about 1.8 million authors and 5 thousand conferences, from 1936 to 2018. We filter out authors that published less than 5 papers and sort the remaining $371,690$ authors by their first publication date to form an author stream. We use this dataset to study the *representative author selection* problem: an author represents a conference if the author published papers in the conference, and our goal is to maintain k authors that jointly represent the maximum number of distinct conferences at any time.

MemeTracker [23]. The dataset contains $714,072$ news and blog articles published in Jan 2009. Each article is represented as a set of memes (such as quotes, phrases, and links). We use this dataset to study the *representative article selection* problem, and the goal is to maintain k articles that jointly cover the maximum number of distinct memes at any time.

math.StackExchange [24]. The dataset records about one million questions posted from July 2010 to June 2018. Each question is represented as a set of tags. We use this dataset to study the *representative question selection* problem, and the goal is to maintain k questions that jointly cover the maximum number of distinct tags at any time.

StackOverflow [24]. We use a larger StackOverflow data to study the same representative question selection problem. The dataset records about 3 million questions posted from Jan 2015 to March 2016. Each question is represented as a set of tags. Our goal is to maintain k questions that jointly cover the maximum number of distinct tags at any time.

Yahoo!-clicks [25]. This dataset records about 4.7 million clicks on web pages on May 1, 2009. Each click is represented as a 5-dimensional feature vector. We use this dataset to study the *Gaussian process active learning* problem (which will be

explained in detail later), and our goal is to maintain k most informative clicks for Gaussian process regression.

NYC-taxi [26]. We also use the New York city yellow taxi trip data to study the same Gaussian process active learning problem. The dataset records about 8.8 million trip records in Jan 2018. Each trip record is represented as a 6-dimensional feature vector. Our goal is to maintain k most informative trips for Gaussian process regression.

The first four problems will be commonly referred to as the *representative item selection* problem. Each input data will be treated as a probabilistic-decaying stream, and we require that the selected core items are biased towards recent data items (cf. Fig. 1(c)). A brief summary of the data is given in Table I.

Table I
STATISTICS OF DATA STREAMS

data stream	item	length	time period
DBLP	author	371,690	1936 - 2018
MemeTracker	article	714,072	1/2009 (one month)
math.StackEx.	question	955,284	7/2010 - 6/2018
StackOverflow	question	2,904,450	1/2015 - 3/2016
Yahoo!-clicks	click features	4,681,992	5/1/2009 (one day)
NYC-taxi	trip features	8,759,874	1/2018 (one month)

B. Settings

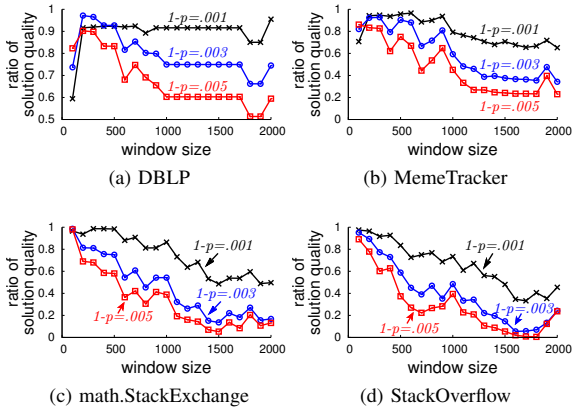
Benchmarks. We will consider the following two baseline algorithms.

- **GREEDY**. A straightforward non-streaming approach to solve CIT over a PDS is to re-run the GREEDY algorithm on $\mathcal{B}_t$ at each time t. GREEDY starts with an empty set $S = \emptyset$, and iteratively, in each step, adds an item s which maximizes the marginal gain, i.e., $s^* \in \arg\max_s \Delta(s|S)$. GREEDY stops once it has selected k items, or the gain becomes zero. To further improve its efficiency, we apply the *lazy evaluation* trick [27]. GREEDY achieves the best $1 - 1/e$ approximation ratio, and will serve as an upper bound of solution quality.
- **Unbiased Reservoir Sampling (Unbiased RS)**. Reservoir sampling (RS) [28] is a single pass streaming algorithm that chooses k samples uniformly at random from a stream. These k samples can be used to estimate the statistical properties of the stream, and thus can be treated as the representative items of the stream. We implemented the reservoir sampling algorithm in [28], referred to as the unbiased RS.
- **Biased Reservoir Sampling (Biased RS)**. Unbiased RS treats past and present items in the stream equally. Aggarwal [29] proposed a temporal biased RS algorithm that tends to choose recent items in the stream as samples. We implemented this RS algorithm and referred to as the biased RS.

Efficiency Measure. We follow the previous work [7] and record the number of utility function evaluations, i.e., number of oracle calls, as a measure of an algorithm's efficiency. The advantage of this measure is that it is independent of the concrete algorithm implementation and platform.

C. Representative Item Selection

Comparing PDS with Sliding-Window Streams. Before evaluating the performance of algorithms for solving the CIT problem, we perform some experiments to compare the PDS with the sliding-window streams with the purpose of answering the question: are core items found on sliding-window streams still good on PDS streams? We vary the window size of a sliding-window stream, and run GREEDY to obtain core items. Then we evaluate the quality of these core items on a PDS, and compare the quality with core items by running GREEDY on the PDS. Let $p_e = p$, $k = 10$, and $n = 50$. The ratios of solution quality are shown in Fig. 9.



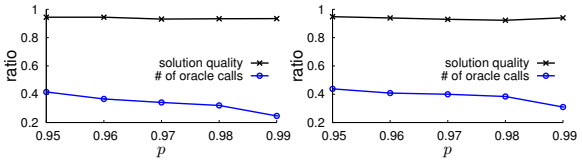
(a) DBLP (b) MemeTracker (c) math.StackExchange (d) StackOverflow

Fig. 9. Comparing PDS with sliding-window streams

In general, we observe that core items found in a small window have higher quality than core items found in a large window, and this trend is significant when the PDS decays fast (i.e., small p). This observation actually shows the different features of PDS and sliding-window streams, i.e., fast decaying PDS emphases recent data items in the stream, and hence core items found in a small window tend to have high quality. But we also observe that when window size is too small, the found core items may have rather poor quality on PDS (see Figs. 9(a) and 9(b)). This shows that sliding-window streams abruptly discard data outside of the window and may result in unstable solution quality because recent but noise data may be chosen as core items.

PDCIT vs. PDCIT+. In this experiment, we compare PDCIT with PDCIT+. The purpose is to study how close their solution quality is, and how significant PDCIT+ can improve the efficiency upon PDCIT. Let $p_e = p \in \{.95, .96, .97, .98, .99\}$, $\epsilon = 0.2$, $k = 10$, $n = 20$, and we truncate lifetimes at $L = 100$. We compute two ratios: (1) *solution quality ratio*, i.e., the quality of solution obtained by PDCIT+ over the quality of solution obtained by PDCIT, and (2) *number of oracle calls ratio*, i.e., the number of oracles of PDCIT+ over the number of oracle calls of PDCIT. Both results are averaged after running 1000 time steps, as shown in Fig. 10.

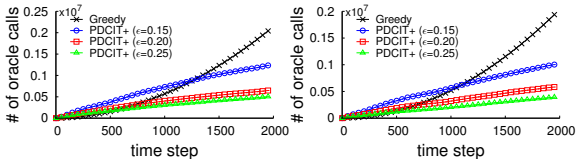
We observe that the solutions found by PDCIT+ are slightly worse than the solutions found by PDCIT, but they are indeed very close: the ratio of solution quality is larger than 0.9.



(a) DBLP (b) MemeTracker

(c) math.StackExchange (d) StackOverflow

Fig. 10. PDCIT vs. PDCIT+



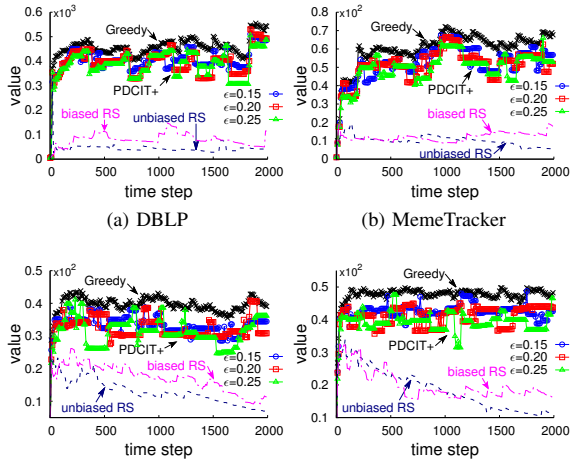
(a) DBLP (b) MemeTracker

(c) math.StackExchange (d) StackOverflow

Fig. 12. Efficiency comparison (lower is better)

PDCIT+ is much more efficient than PDCIT, and it can reduce more than a half of the number of oracle calls of PDCIT. This experiment thus demonstrates that PDCIT+ finds solutions with quality close to PDCIT, and PDCIT+ is much more efficient than PDCIT.

Solution Quality. Next, we validate the quality of solutions found by PDCIT+. Let $p_e = 0.999$, $n = 20$, and $k = 10$. We run GREEDY, unbiased RS, biased RS, and PDCIT+ for $2,000$ time steps, respectively, and show the results in Fig. 11.



(a) DBLP (b) MemeTracker

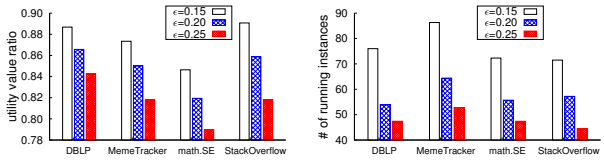
(c) math.StackExchange (d) StackOverflow

Fig. 11. Solution quality comparison (higher is better)

We observe that GREEDY finds the best solutions among the four. RS methods perform poorly, and biased RS is slightly better than unbiased RS. These results are expected. PDCIT+ can find solutions with quality close to GREEDY, and performs much better than RS methods. In general, small ϵ can further improve the solution quality, as shown in Fig. 13(a).

Scalability. We validate the scalability of PDCIT+. Parameter settings in this experiment are the same as in previous experiment. To evaluate efficiency, we record the cumulative number of oracle calls of each algorithm over time, and show the results in Fig. 12.

We observe that GREEDY is only efficient at the very beginning of the stream (when the number of items is small), but the number of cumulative oracle calls increases very quickly over time. In contrast, the number of oracle calls of PDCIT+ grows relatively slow. This observation indicates that PDCIT+ is indeed much more efficient than GREEDY. In general, large ϵ can further improve the efficiency. This is because the number of running MC-SIEVESTREAMING instances in PDCIT+ decreases, as shown in Fig. 13(b).



(a) average utility value ratio w.r.t. GREEDY (b) average number of running instances, i.e., $|\mathbf{x}_t|$

Fig. 13. Trade-off effect of parameter ϵ

D. Gaussian Process Active Learning

Our technique can also be used to improve the efficiency of some machine learning tasks such as Gaussian process regression [30,31]. Gaussian process regression is a non-parametric Bayesian approach towards regression problems, and the goal is to learn a function mapping inputs (e.g., predictor variables) to outputs (e.g., target variables). Gaussian process is an expressive method to model unknown functions. It uses a *kernel function* K to transform points in a non-linear space to a typically high-dimensional linear space, for which good algorithms are available. One of the frequently used kernel functions is the *radial basis kernel*, i.e.,

$$K(e_i, e_j) \triangleq \sigma^2 \exp(-\frac{\|e_i - e_j\|^2}{2\lambda^2}) \tag{10}$$

for two input points e_i and e_j. Here, σ^2 (the *signal variance*) and λ (the *length scale*) are two hyper-parameters.

In Gaussian process active learning, we want to actively choose the input points for which to observe the outputs in

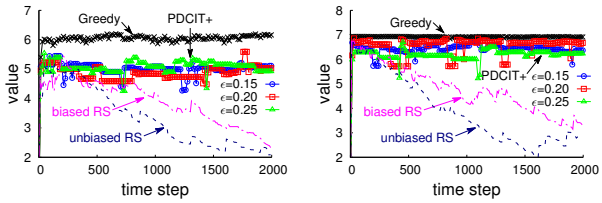
order to learn the unknown function as quickly and accurately as possible. Under the *information gain* criteria (i.e., maximize the reduction in uncertainty), the objective becomes to choose observation points S to maximize the utility function

$$f(S) = \frac{1}{2} \log \det \left(I + \sigma^{-2} K(S, S) \right) \tag{11}$$

where $K(S, S)$ is a $|S| \times |S|$ kernel matrix with each element defined by Eq. (10). $f(S)$ is proved to be monotone and submodular [31].

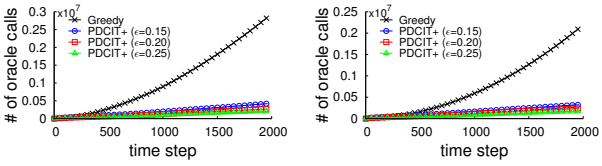
Gaussian process active learning on large scale dataset is notoriously inefficient. Our technique can be used to choose informative training points in a streaming fashion efficiently. Importantly, our technique allows to smoothly forget outdated training points, which is important in keeping the model fresh.

We apply our technique to Yahoo!-clicks and NYC-taxi datasets to choose k most informative training points, respectively. Let $\sigma = 1, \lambda = 0.5, k = 10, n = 20, p_e = 0.999$. We run different algorithms for $2,000$ time steps as before. The results are shown in Figs. 14 and 15.



(a) Yahoo!-clicks (b) NYC-taxi

Fig. 14. Solution quality comparison (higher is better)



(a) Yahoo!-clicks (b) NYC-taxi

Fig. 15. Efficiency comparison (lower is better)

These observations all consist with our previous results. PDCIT+ can find solutions with quality comparable with GREEDY (within 80%), and PDCIT+ performs much better than RS methods. PDCIT+ also uses much less oracle calls than GREEDY. The efficiency is improved to about one order of magnitude faster.

VIII. RELATED WORK

We briefly review some related literature in this section.

Stream Sampling. Another widely used technique to reduce the scale of large data streams is stream sampling such as reservoir sampling [28,29] and distinct sampling [32,33]. Note that the goal of stream sampling is to select a few data items as *samples* to approximate the statistical properties of original data stream. While in CIT, our goal is to find an optimal set of data items as core items to maximize a utility function, which can measure the informativeness or representativeness of a set of data items. If the data stream contains too many redundant or noise items, stream sampling may result in many useless samples [32].

Submodular Optimization. Submodular function maximization [34] has been extensively studied in the past few years due to its wide applications in influence maximization [14], sensor placement [4,35], search result diversification [13], feature selection [10], data summarization [36]–[38], and so on. For cardinality constrained monotone submodular function maximization, the classic GREEDY algorithm guarantees a $(1 - 1/e)$ approximation ratio [17]. To further improve the efficiency of GREEDY, many techniques have been proposed, e.g., lazy evaluation [4,27], distributed computation [39]–[41], sampling [42], etc. Different from these existing approaches, our work belongs to a category of streaming algorithms where each incoming data item can only be accessed once.

Streaming Submodular Optimization (SSO). Our approach belongs to a kind of SSO techniques, which are efficient methods to handle massive streaming data using sublinear memory and update time. Existing SSO techniques can only handle insertion-only streams [3,7] and sliding-window streams [9,18,19]. For insertion-only streams, the state-of-the-art streaming algorithm is SIEVESTREAMING [7], which guarantees an $(1/2 - \epsilon)$ approximation ratio. For sliding-window streams, the state-of-the-art streaming algorithm is proposed by Epasto [19], which guarantees a $(1/3-\epsilon)$ approximation ration. (Note that approximation ratio $(1/2-\epsilon)$ is also achievable but requires higher computational complexity.) As we pointed out in introduction, these two existing data stream modes represent two extremes: the insertion-only streams cannot satisfy the recency requirement; while the sliding-window streams cannot satisfy the continuity requirement. Hence, the SSO techniques built on them will also suffer some weaknesses. Recently, [43] designed a new SSO technique for streams with inhomogeneous decays. That is, each item in the stream can participate in analysis for an arbitrary amount of time. Such a model slightly generalizes these two extreme stream models, however, it is still not clear how to determine each item's participation time in practice. Instead, our proposed PDS model is more elegant to continuously handle evolving data streams.

Maintaining Time-Decaying Stream Aggregates. Cohen et al. [44] first extend the sliding-window model in [45] to the general time-decaying model for the purpose of approximating summation aggregates in data streams (e.g., estimating the number of 1's in a 0-1 stream). Cormode et al. [46] consider the similar problem by designing time-decaying sketches. These studies have inspired us to consider the more general probabilistic-decaying streams.

IX. CONCLUSION

This work provides a tool to help people reduce data overload and draw valuable insights from evolving data streams. We formulate our problem as a core items tracking (CIT)

task over probabilistic-decaying streams. This setting allows us to gradually forget outdated data, and it is more elegant and general than the insertion-only and sliding-window streams, which represent two extremes. We address the CIT task by designing new SSO techniques: PNDCIT solves CIT over a special kind of probabilistic non-decaying streams, and guarantees an $(1/2 - \epsilon)$ approximation ratio; PDCIT leverages PNDCIT as a building block to solve CIT over general probabilistic-decaying streams, and also guarantees an $(1/2-\epsilon)$ approximation ratio; PDCIT+ improves the efficiency of PDCIT a lot, and guarantees a $(1/4 - \epsilon)$ approximation ratio. Our proposed techniques can find solutions with quality comparable with GREEDY but improve the computational efficiency to about one order of magnitude faster.

ACKNOWLEDGMENT

The research presented in this paper is supported in part by National Natural Science Foundation of China (61902305, 61922067, U1736205), Shenzhen Basic Research Grant (JCYJ20170816100819428), MoE-CMCC "Artifical Intelligence" Project (MCM20190701), Natural Science Basic Research Plan in Shaanxi Province of China (2019JM-159), Natural Science Basic Research Plan in ZheJiang Province of China (LGG18F020016).

REFERENCES

[1] D. Chakrabarti and K. Punera, "Event summarization using tweets," in *ICWSM*, 2011.
[2] H. Zhuang, R. Rahman, X. Hu, T. Guo, P. Hui, and K. Aberer, "Data summarization with social contexts," in *CIKM*, 2016.
[3] B. Saha and L. Getoor, "On maximum coverage in the streaming model and application to multi-topic blog-watch," in *SDM*, 2008.
[4] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, and N. Glance, "Cost-effective outbreak detection in networks," in *SIGKDD*, 2007.
[5] G. Cormode, S. Muthukrishnan, and W. Zhuang, "Conquering the divide: Continuous clustering of distributed data streams," in *IEEE ICDE*, 2007.
[6] C. Patil and I. Baidari, "Estimating the optimal number of clusters k in a dataset using data depth," *Data Science and Engineering*, vol. 4, no. 2, pp. 132–140, 2019.
[7] A. Badanidiyuru, B. Mirzasoleiman, A. Karbasi, and A. Krause, "Streaming submodular maximization: Massive data summarization on the fly," in *SIGKDD*, 2014.
[8] J. Xu, D. V. Kalashnikov, and S. Mehrotra, "Efficient summarization framework for multi-attribute uncertain data," in *SIGMOD*, 2014.
[9] Y. Wang, Y. Li, and K.-L. Tan, "A sliding-window framework for representative subset selection," in *IEEE ICDE*, 2018.
[10] G. Brown, A. Pocock, M.-J. Zhao, and M. Luján, "Conditional likelihood maximisation: A unifying framework for information theoretic feature selection," *Journal of Machine Learning Research*, vol. 13, pp. 27–66, 2012.
[11] R. Mehrotra and E. Yilmaz, "Representative & informative query selection for learning to rank using submodular functions," in *SIGIR*, 2015.
[12] M. Babaei, P. Grabowicz, I. Valera, K. P. Gummadi, and M. Gomez-Rodriguez, "On the efficiency of the information networks in social media," in *WSDM*, 2016.
[13] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong, "Diversifying search results," in *WSDM*, 2009.
[14] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *SIGKDD*, 2003.
[15] P. Indyk, S. Mahabadi, M. Mahdian, and V. S. Mirrokni, "Composable core-sets for diversity and coverage maximization," in *PODS*, 2014.
[16] S. Stergiou and K. Tsioutsiouliklis, "Set cover at web scale," in *SIGKDD*, 2015.
[17] G. Nemhauser, L. Wolsey, and M. Fisher, "An analysis of approximations for maximizing submodular set functions - I," *Mathematical Programming*, vol. 14, pp. 265–294, 1978.
[18] J. Chen, H. L. Nguyen, and Q. Zhang, "Submodular maximization over sliding windows," in *arXiv:1611.00129*, 2016.
[19] A. Epasto, S. Lattanzi, S. Vassilvitskii, and M. Zadimoghaddam, "Submodular optimization over sliding windows," in *WWW*, 2017.
[20] K. U. Khan, W. Nawaz, and Y.-K. Lee, "Set-based unified approach for summarization of a multi-attributed graph," *World Wide Web*, vol. 20, pp. 543–570, 2017.
[21] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*, 2nd ed. Springer, 2004.
[22] "DBLP computer science bibliography," http://dblp.dagstuhl.de, 2019.
[23] "MemeTracker data," http://www.memetracker.org/data.html, 2019.
[24] "StackExchange data," https://archive.org/details/stackexchange, 2019.
[25] "Yahoo! front page today module user click log dataset, version 1.0," https://webscope.sandbox.yahoo.com, 2019.
[26] "TLC trip record data," https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2019.
[27] M. Minoux, "Accelerated greedy algorithms for maximizing submodular set functions," *Optimization Techniques*, vol. 7, pp. 234–243, 1978.
[28] J. S. Vitter, "Random sampling with a reservoir," *ACM Transaction on Mathematical Software*, vol. 11, no. 1, pp. 37–57, 1985.
[29] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *VLDB*, 2006.
[30] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, 1st ed. The MIT Press, 2006.
[31] E. Schulz, M. Speekenbrink, and A. Krause, "A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions," *Journal of Mathematical Psychology*, vol. 85, pp. 1–16, 2018.
[32] J. Chen and Q. Zhang, "Distinct sampling on streaming data with near-duplicates," in *PODS*, 2018.
[33] P. Wang, X. Wang, J. Tao, P. Zhang, and X. Guan, "Continuously distinct sampling over centralized and distributed high speed data streams," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 300–314, 2019.
[34] A. Krause and D. Golovin, *Submodular Function Maximization*. Cambridge University Press, 2014, pp. 71–104.
[35] A. Krause, C. Guestrin, A. Gupta, and J. M. Kleinberg, "Robust sensor placements at informative and communication-efficient locations," *Journal ACM Transactions on Sensor Networks*, vol. 7, no. 4, 2011.
[36] B. Mirzasoleiman, S. Jegelka, and A. Krause, "Streaming non-monotone submodular maximization: Personalized video summarization on the fly," in *AAAI*, 2018.
[37] M. Mitrovic, E. Kazemi, M. Zadimoghaddam, and A. Karbasi, "Data summarization at scale: A two-stage submodular approach," in *ICML*, 2018.
[38] B. Mirzasoleiman, A. Karbasi, and A. Krause, "Deletion-robust submodular maximization: Data summarization with the right to be forgotten," in *ICML*, 2017.
[39] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause, "Distributed submodular maximization," *Journal of Machine Learning Research*, vol. 16, pp. 1–41, 2015.
[40] A. Epasto, V. Mirrokni, and M. Zadimoghaddam, "Bicriteria distributed submodular maximization in a few rounds," in *SPAA*, 2017.
[41] R. Kumar, B. Moseley, S. Vassilvitskii, and A. Vattani, "Fast greedy algorithms in MapReduce and streaming," in *SPAA*, 2013.
[42] B. Mirzasoleiman, A. Badanidiyuru, A. Karbasi, J. Vondrak, and A. Krause, "Lazier than lazy greedy," in *AAAI*, 2015.
[43] J. Zhao, S. Shang, P. Wang, J. C. Lui, and X. Zhang, "Submodular optimization over streams with inhomogeneous decays," in *AAAI*, 2019.
[44] E. Cohen and M. J. Strauss, "Maintaining time-decaying stream aggregates," *Journal of Algorithms*, vol. 59, pp. 19–36, 2006.
[45] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1794–1813, 2002.
[46] G. Cormode, S. Tirthapura, and B. Xu, "Time-decaying sketches for robust aggregation of sensor data," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1309–1339, 2009.